ARO 18012.1-S-EL

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. AD-A109 136 |
| | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) STUDY AND EXPERIMENTATION OF HORIZONTAL MICROPROCESSOR MACHINE DESCRIPTION TECHNIQUES | 5. TYPE OF REPORT & PERIOD COVERED Final Report 4/81 - 11/81 |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) John L. Gieser, Robert J. Sheraga | 8. CONTRACT OR GRANT NUMBER(s) DAAG29-81-C-0018 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS JRS Industries, Inc. 11722 Sorrento Valley Road San Diego, Ca 92121 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709 | 12. REPORT DATE 1 November 1981 |
| | 13. NUMBER OF PAGES 82 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | 15. SECURITY CLASS. (of this report) Unclassified |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.



17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Micro-architectures, Micro-engine description techniques, Automatic microcode generation, High level microprogramming languages.



20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document reports on the research and investigation aspects of micro-architecture description methodologies. Its primary objectives are: 1) to identify the methodology that appears to have the most promise for use in an automatic microcode generation system; and 2) to define the means for developing the necessary tools and techniques needed for further evaluating this methodology. In automatically generating microcode from a high level source language, a significant issue is the description of the target micro-

**JRS INDUSTRIES, INC.**
TECHNICAL SERVICES DIVISION
11722 SORRENTO VALLEY ROAD, SAN DIEGO, CALIFORNIA 92121 / TELEPHONE (714) 755-4072

# STUDY AND EXPERIMENTATION OF HORIZONTAL MICROPROCESSOR MACHINE DESCRIPTION TECHNIQUES

FINAL REPORT

AUTHORS:

JOHN L. GIESER
ROBERT J. SHERAGA

1 NOVEMBER 1981

U.S. ARMY RESEARCH OFFICE

CONTRACT NUMBER: DAAG29-81-C-0018

JRS

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHOR(S) AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

JRS

FOREWORD

This document is provided to the U.S. Army, Army Research Office, in fulfillment of part of the documentation requirements associated with Contract No. DAAG29-81-C-0018, 'Study and Experimentation of Horizontal Microprocessor Machine Description Techniques.' The purpose of this document is to provide a summary of all work performed under the contract along with conclusions and recommendations.

This document reports on the research and investigation aspects of microarchitecture description methodologies. Its primary objectives are: 1) to identify the methodology that appears to have the most promise for use in an automatic microcode generation system and 2) to define the means for developing the necessary tools and techniques and experimentally evaluating the methodology. Implementation of and experimentation with the methodology will necessitate future efforts and activities using this document as a reference and point of departure.

JRS

TABLE OF CONTENTS

JRS

## LIST OF FIGURES

# 1.0 INTRODUCTION

JRS Industries, Inc. (JRS) is conducting research and experimentation in technology areas associated with horizontally microprogrammed processors. Key areas of activity include the use of such devices as processing nodes in a distributed computer network, the automatic generation of efficient microcode for such devices starting with application programs written in a high level language (HLL), and the customization of machine architectures for the solution of specific application problems.

JRS has developed a significant base of tools and techniques which are exceptionally valuable in conducting research and experimentation in these areas. A software system, that compiles HLL programs and generates efficient microcode for horizontally microprogrammed processors, has been designed and implemented. The system includes several features that are useful for this experimental activity, including schemas for concurrency detection and code compaction. Additionally, the system allows for the introduction of the architectural description of a target machine into the microcode generation process in a manner that facilitates experimentation with machine architectures and their relationship to application problems.

A generalized machine description capability, that deals with the issues peculiar to generating microcode for horizontally microprogrammed processors, is clearly needed. It is fundamental to the basic problem of automatic microcode generation and it is basic to the practical consideration of using horizontal machines as nodes in a distributed network, particularly if each node may be unique; additionally, it is a tool that is required for the conducting of efficient research and experimentation in this field.

This document presents the results of a study and experiment effort to design, develop, and evaluate techniques and tools necessary to parameterize high performance, horizontally microprogrammed processors. The effort has as its primary goal the development of a practical, generalized, and portable procedure which would enable a person, who is thoroughly familiar with a processor, to extract from the hardware schematics and timing diagrams, and express clearly and concisely, the structural and behavioral features of the machine which are relevant to efficient microcode generation. This information would then be used to prepare input that could be referenced by a generalized microcode generator to create microcode for that target machine. In order to be feasible, the code

produced using this process must be usable and efficient, and the use of these techniques and tools must be both cost effective and efficient in terms of schedule and operation.

The benefits that result from achieving this primary goal are many. The procedure would have value simply as a tool for describing a horizontal processor clearly and concisely. It would permit the target processor to be changed rather easily in the existing automatic microcode generation system, thereby enabling experimental code production for a variety of machines. Of particular significance is that the procedure would enable one to hypothesize a microarchitecture with specific characteristics and to generate microcode for a particular application problem that will run on this microengine. One would then experiment by making modifications to the architecture and observing the resulting effects on the microcode produced. This capability would enable the identification of desireable architectural features of machines for specific applications in a very direct and measurable way.

## 2.0 PROBLEM OVERVIEW

The overall problem to be solved is that of developing the techniques and tools necessary to enable the efficient production of microcode for horizontally microprogrammed processors. One of the approaches deemed promising for the solution of this problem is the use of a high level language (HLL), which is oriented toward the system problem, for coding of the application software and the subsequent compilation and transcription of HLL statements into microcode for the horizontal processor.

The following description is intended to highlight and summarize the overall process under study and experimentation; that is, the process of transforming statements written in a HLL format into a sequence of statements executable in the target machine format. The description is given sequentially with a graphic depiction of the process included as Figure 2-1. The reader is referred to Appendix A for a more detailed treatment of the process.

Programs for performing system information processing (application programs) are prepared using a HLL. The HLL source statements are "compiled", generating sequences of procedural operations. These operations define an intermediate language (IL1) which will convey the semantic structure of the original HLL source statements. Additionally, IL1 is more specific in designating machine resources and is highly symbolic and compacted to allow for easier subsequent manipulation.

An analysis is made of the IL1 statement sequences to determine their concurrency of operations and timing dependencies. This time-and-data partitioning of the IL1 statement input stream will allow the later code generation step to maximize the amount of parallel processing performed up to the limits imposed by the application problem being solved and the processing capabilities of the target machine. The output of this step is Intermediate Segments of Straight Line Code (ISLC) and their associated time dependency limitations. The ISLC segments described above undergo a translation to an additional intermediate Language (IL2) to generate sequences of primitive operations. The IL2 expressed operation sequences generated are in a symbolic format and must again be analyzed for resource and time-and-data dependencies.

At this stage of the compilation process, the symbolic primitive operations and symbolic resource designators are transformed into machine dependent micro-operation sequences (Intermediate Language 3, IL3) through the use of a microarchitecture description model and an IL2 instruction set interpretation scheme. A detailed description of the

techniques used in microarchitecture description and instruction set interpretation are the object of this report and are covered in detail in the remainder of this document.

It is also at this stage that the code generation and improvement schemas are used. The goal of this step is to compact micro-operations into micro-instructions to the maximal extent possible, thereby effecting a high degree of parallelism. To generate efficient microcode also requires the ability to access and best-match the current state of the microengine's resources. The issues of instruction set interpretation and code generation are closely bound to the general resource allocation scheme used by the code generator.

One of the goals of this study and experiment is to retain machine independence and portability in as many of the tools and techniques developed as possible without bearing a significant economic impact on their use in the compilation process. Furthermore, since there are many machine architectures that would be useful to experiment with, it is necessary that the machine description procedure be generalized and made applicable to as wide a range of architectures as possible. Moreover, the procedure must be relatively easy to use, automated to the extent practical, and integrated with the existing experimental software system. In short, if the existing experimentation system can be made easily retargetable, (i.e., able to generate code for more than one target machine), then the range of problems to which the tool may be usefully applied will increase significantly and the effectiveness and efficiency of the research will be greatly enhanced.

The most significant test data used for the study and experiment are presented in Section 3.3 of this document.

Section 4.0 of this document presents the conclusions and recommendations resulting from the study and experiment effort. Also included in that section are some of the more significant assumptions made during the course of the project.

JRS

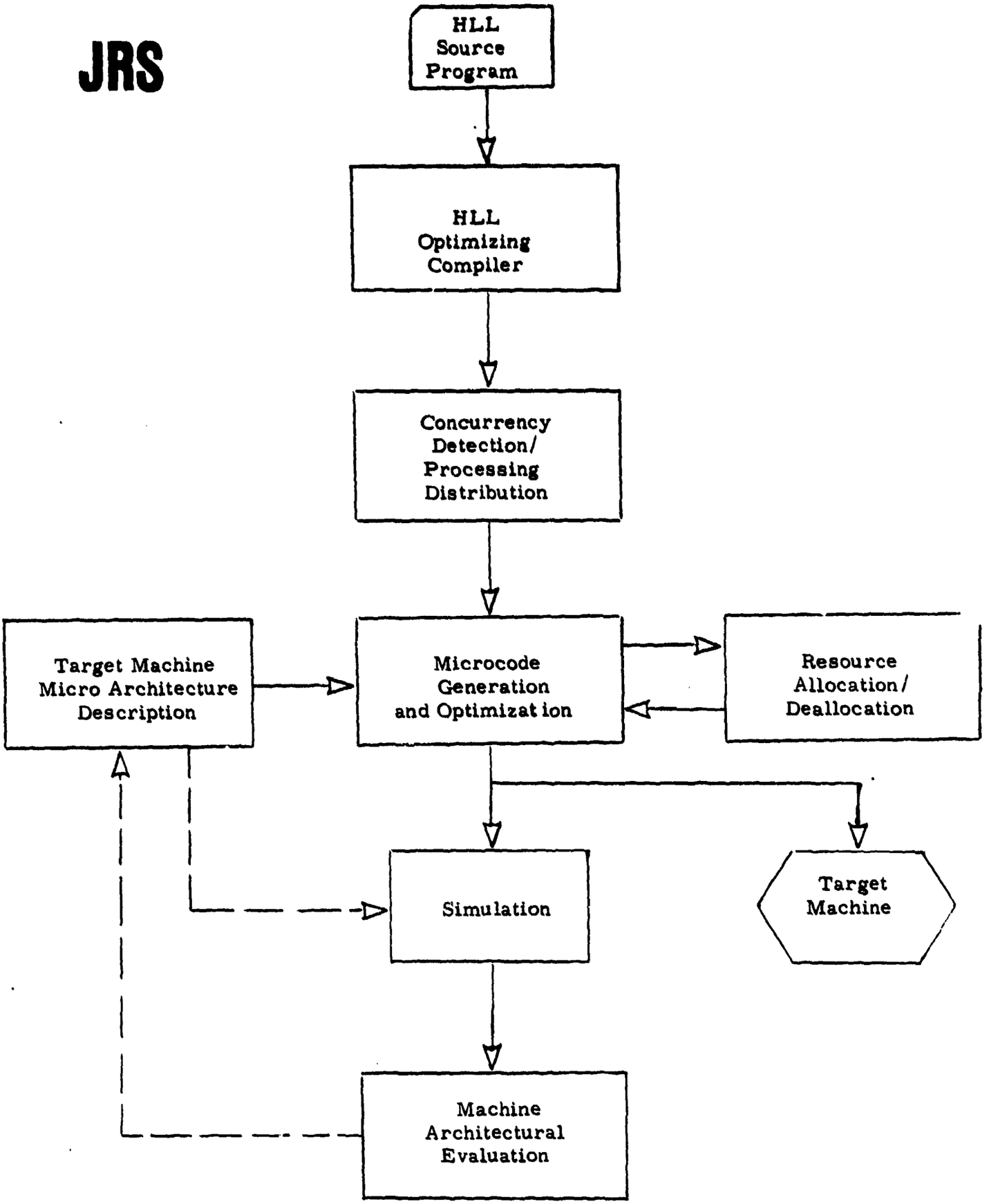



Figure 2-1. HLL-TO-MICROCODE COMPILATION SYSTEM

## 3.0 STUDY RESULTS

The overall problem associated with a generalized microarchitecture description procedure may be divided into two major steps. First, one must devise a method for collecting and expressing the structural and behavioral features of a microengine; and secondly, one must establish the approach for introducing this descriptive information into the compilation process. The method devised in the first step is referred to as the Microarchitecture Description Methodology (MDM). The second step approach is referred to as Instruction Set Interpretation (ISI).

The description methodology presented in this document is not to be confused with the use of the terminologies "Computer Hardware Description Languages", "Machine Description Languages", etc. The techniques and procedures presented here are intended only to allow for the extraction of the information required in the HLL-to-Microcode compilation process for a specific target machine. The latter terminology refers to the methodologies and procedures used in machine architecture description and design for creation and development purposes. Additionally, the latter typically refers to "macro" level computer characteristics as opposed to a description of the structural and behavioral features of a processor's micro-engine.

## 3.1 MICROARCHITECTURE DESCRIPTION METHODOLOGY

The description methodology covers the following four basis areas:

- Microinstruction description
- Element descriptions
- Microoperation usage rules
- Micro engine behavioral rules

The microarchitecture description methodology (MDM) is prefaced with definition, convention, interpretation and nomenclature standards to be applied to insure consistency and uniformity in these basic areas.

An overview of the MDM procedure is given in Figure 3-1. In the figure, note that 'Diagrams' in the input area (top of page) is used to encompass the documentation discussed below.

Each of the four basic areas, as well as source document requirements, is discussed in more detail in what follows.

A parameter of significant importance in a microengine description is timing. Timing is used to mean those instances or time periods when events or actions occur or are occurring. The ability to specify detailed timing in a micro-architecture is paramount to generating efficient microcode. Timing refers not only to the sequence of execution of a microinstruction, but also to resources directly and indirectly affected by the execution of a microinstruction.

In the MDM, the timings of interest are the following:

$t_{MOP}$ - the time to select and activate the resources of a MOP

$t_A$ - the time for which the resulting action of the executing MOP is valid

$t_R$ - the time for which a selected resource is busy

The times $t_{MOP}$ and usually $t_A$ are within a microcycle, while the times $t_R$ are typically more than one cycle. As an example, the MOP which specifies a read from large memory using the memory address register, MAR, is activated during time, $t_{MOP}$, within a micro cycle. $t_A$ is typically the time from MOP selection to the end of the current micro-

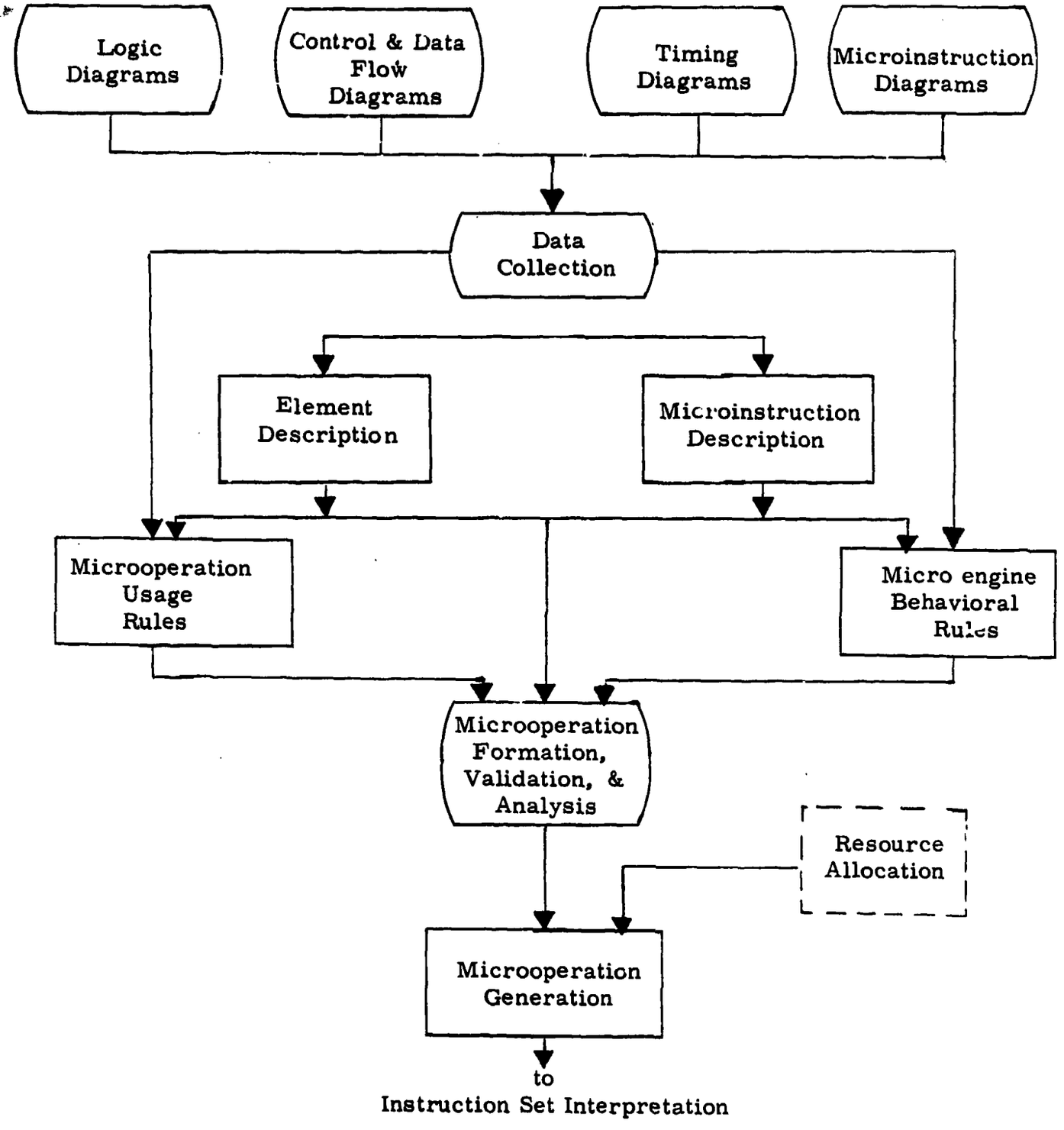



Figure 3-1. Microarchitecture Description Methodology

cycle so that $t_{R(MAR)} > 1$ cycle and the large memory is busy for several cycles so that $t_{R(MEMORY)} >> 1$ cycle. The time, $t_{MOP}$, can be further refined and classified as follows:

$t_{MOP_S}$ - the time within the cycle at which the MOP resources are selected (usually denoted $t_S$).

$t_{MOP_D}$ - the time within the cycle at which the MOP resources are deselected, i.e., completed. (unusally denoted $t_D$)

$t_{MOP_A}$ - the time duration for MOP execution. Thus,

$$t_{MOP_A} = t_{MOP_D} - t_{MOP_S}$$

Similarily the time, $t_A$, can be further classified based on the duration of the action (signals) generated by the executed MOP. This classification of actions is:

dynamic - the action perseveres for one complete cycle

transient - the action perseveres until the end of the current cycle

latched - the action is saved (stored) by the end of the current cycle

static - the action perseveres until reset

The times, $t_A$, have a significant affect on code efficiency. In particular, the times, $t_A$, determine the flexibility available in selecting MOPs and the ability to combine and compact MOPs within a microinstruction (i.e., the degree of concurrency acheivable). Furthermore, when $t_A$ is greater than 1 cycle time, the interaction amongst MOPs (behavioral features) becomes a severely limiting factor in code compaction due to the need to schedule the required resources.

The scheduling of resources is even more critical when considering those resources for which $t_R >> 1$. It is for those resources with large $t_R$ that resource allocation/deallocation techniques can make a significant contribution to the efficiency of the code produced.

### 3.1.1 Source Documents

The methodology assumes the existence and availability of the following documentation on the micro engine and its interfaces:

- Logic diagrams, along with a detailed technical description
- Control and data flow diagrams and their detailed technical descriptions
- Timing diagrams and layouts, with a detailed technical description
- Microinstruction formats and content, along with a detailed technical description

This documentation is assumed to exist at some level of detail; it is intuitively felt that the lower the level of detail (in the sense of more exact and specific details), the better the microarchitecture description will be (in the sense of potentially better utilization of micro engine features and potentially more efficient microcode generation). It is also felt that, as a result of the organizational structure of the collected micro engine data, the methodology will continue to have descriptive value even though less and less detail is available for the micro engine.

The documentation mentioned above is meant to cover the topics typically addressed in a good functional and/or operational specification. The breakout given was used solely for the purposes of identifying and specifying document content and required information. By way of example and definition, the following gives some of the more significant kinds of information expected in the documentation using the perhaps artifical groupings listed above.

In the logic diagrams group should be details of all major subsystems within the micro-engine along with their interconnects. Additionally the salient characteristics of these subsystems and interconnects would be given. In general, this documentation would detail all significant functional elements of the micro engine and how they relate.

Control and data flow documentation should detail all the allowable data and control paths amongst the microengine elements. It should specify

path size and transmission characteristics. Additionally this documentation should detail all available control mechanisms for data and control paths as well as for all the other hardware elements.

Timing diagrams and timing layout documentation should detail the clocking and phasing of the microengine. In particular, it should detail element selection and activation times as well as signal duration times. This documentation should relate the timings of elements, one-to-another, preferably through the use of timing sequence layouts.

The microinstruction format and content documentation should detail all fields in the microinstruction in terms of their size, content, and relationship, one-to-another. It should detail the operation of any hardware-assisted micro-operations, the operational characteristics for use of multi-functional fields, if any, and details of the branching capability available in the microinstruction. Of particular interest are details of the execution ordering (sequencing) and execution concurrency or lack thereof for microoperations.

### 3.1.2 Microinstruction Description

The microinstruction format and content description area is used to present, in detail, the possible microinstruction formats, the fields within a microinstruction format, and to assign symbols to the numeric values of each field. These symbols provide a syntactical link to other sections of the MDM and provide a mechanism for expressing micro-operations (MOPs) symbolically. Comments may be used freely to explain fields and values as needed; however, they are not considered a part of the description. The microinstruction description is accomplished using a series of keywords to describe each control word format and the attributes of each field within the format. This description could be performed eventually through an interactive dialog at a computer terminal, with the computer using the keywords as prompts for user inputs.

Different "formats" of the instruction word are defined to exist when the value in one or more fields selects one of a set of interpretations for the remaining bits of the instruction. Different formats may involve completely different field layouts in each format. In contrast, the situation in which one or more bits within a field governs the interpretation of the remaining bits of that field only, is considered to be simply an encoded field representation in a single instruction format.

Bit length specifications are, by convention, expressed in the form:

$$\langle n{:}m \rangle$$

where n is the leftmost, and m the rightmost bit of the field, numbered according to the conventions specified for the machine. Similarly, timing specifications are expressed as:

$$\langle T_S, T_D \rangle$$

where $T_S$ is the selection time and $T_D$ the deselection time for the element. Conventions will be established to permit $T_S$ and $T_D$ to be expressed in either absolute or relative terms, and to permit $T_D$ to represent a time which exceeds one instruction cycle. The timing specification is an attempt to further refine the polyphase instruction timing concept. The more precisely the activation times of elements can be specified, the more efficiently the MOPs can be combined. Thus, knowing that the ALU is active for a specific operation from 50 to 95 ns within the cycle, for example, may lead to better code compaction than knowing it is active during "phase 3".

# JRS

The microinstruction definition proceeds by specifying information about the entire control word, then each format in turn. Each field and its values are then defined within each format. It is assumed that all formats of the instruction are of the same length, but not necessarily that they have the same execution time. A sampling of the keywords used in the definition is given below.

```
MICROINSTRUCTION: instruction_name
   BITS: <n:m>                          ; Instruction width in bits
   FORMATS: number                      ; Number of formats in this
                                          instruction
   DEFAULT FORMAT: format_name          ; Default type

FORMAT #j: format_name                  ; Repeated for each format
   FIELDS: number                       ;  Number of fields in this format
   TIMING: <0, T_C>                     ;  Cycle time for format #j


FIELD #k: field_name                    ; Repeated for each field
   BITS: <n:m>
   TYPE:                                ; Field type
       DATA_ADDRESS  }  Direct          ;  Memory address
       INST_ADDRESS  }  Value           ;  Instruction address
       LITERAL       }                  ;  Literal value
       SELECTOR      }                  ;  Selects one of a set of identical
                     }  Encoded            components
       SWITCH        }  Value           ;  Multiplexor or enable/disable
                     }                     settings
       CONTROL       }                  ;  Encoded control functions
   VALUES:                              ; For encoded values
       0...0:value_name                 ;  Value-name is a unique symbol
          .                             ;  used to represent each allowable
          .                             ;  value of the field
          .                             ;
       1...1:value_name                 ;
   DEFAULT_VALUE:                       ; Default value for this field
       Direct value or value_name
```

The microinstruction definition is probably the most straightforward to obtain from typical machine documentation. The format used is similar to that for specifying field names in a generalized microassembler. The utility of this information is that it permits symbolic names to be attached to fields and values, which serve to link together the segments of the architecture description.

### 3.1.3 Element Descriptions

The elements of the machine are the [hardware] components including storage units (memories, registers, latches, etc.) and transformation units (ALU's, shifters, etc.). It is also a useful convenience to define pseudo-elements, which are conceptual elements (normally registers) used to simplify the descriptions of component inputs and outputs.

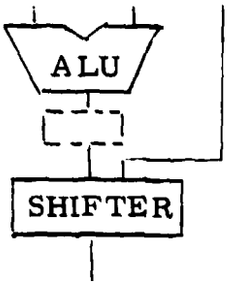
In general terms, there will be an element description for each component shown on a detailed block diagram of the machine architecture. Certainly this is true of storage and transformation units. Link units, such as buses, generally will not require definition, but a pseudo-element may be needed if a link has any function other than an ordinary data path. Multiplexors are defined as elements.

Element descriptions consist of specifications of the inputs and outputs for each element and the timing of the element within the cycle. In addition, for transformation units, the functions performed by the unit must be given. For storage units, particularly registers, it is convenient to give the specifications for a set of identical components as a group rather than to specify each element individually.

The list of element descriptions should be closed, and internally consistent. Each input and output must be the name of an element in the list as a field name. All elements must have at least one input and one output. The use of conceptual pseudo-elements helps to make this possible in a logical manner.

Pseudo elements of two types are used:

1. Storage pseudo-elements function like registers. Their use makes it possible to isolate elements which would otherwise have to be treated as a single larger unit. For example, if the ALU-shifter configuration in the machine is:



Then the definition of a storage pseudo-element ALUOUT which can be thought of as a register which holds the output of the ALU and becomes an input to the shifter, permits the ALU and shifter to be isolated as separate elements.

2. Transformation pseudo-elements permit the description of operations which take place below the level of detail normally specified in architecture diagrams; for example, autoincrementing registers or split data paths. These pseudo-elements normally do correspond to some hardware components in the machine, which are not microprogrammable. The definition of a transformation pseudo-element is normally dictated by a data path in the machine which performs something other than an identity data transfer function.

Parts of the element descriptions require further definition. It is not yet clear whether it will be useful to establish classes, such as general registers, control registers, scratchpad registers, and external registers; or whether some standard, pre-defined names should be used for certain elements, such as MAR, MDR (memory address register, memory data register). It may be useful to conceive of all elements as transformation units; and include the identity and null functions as legal transformations.

A sample of the keywords, format, and syntax used in the element description section is given below. Figure 3-2 gives examples of the most commonly used storage element descriptions, while Figures 3-3 and 3-4 give these examples for link elements and transformation elements, respectively. Figure 3-5 presents a rather complete syntax for transformation element operations. Section 3.1.3.1 below gives the semantics for the transformation element operations.

```
REGISTER : stor_elem_name
   INPUT : elem_name , field_name,...
   OUTPUT : elem_name,...
   BITS : < n:m>
   READ_TIME : < T_S, T_D >
   WRITE_TIME : < T_S, T_D>

REGISTER_SET : stor_elem_name
   INPUT : elem_name, field_name,...
   OUTPUT : elem_name,...
   BITS : < n:m>
   READ_TIME : <T_S, T_D >
   WRITE_TIME : < T_S, T_D>
   NUMBER : n

DISCRETE : discrete_name
   SET_BY : tran_elem_name, field_name,...
   USED_BY : tran_elem_name,...
   READ_TIME : <T_S, T_D >
   WRITE_TIME : < T_S, T_D>
   TYPE :                        ; When contents are changed
      LATCHED                    ; Contents stored and saved
      DYNAMIC                    ; Contents preserved for 1 cycle time
      TRANSIENT                  ; Contents preserved until end of current cycle
      STATIC                     ; Contents preserved until reset
PSEUDO_REGISTER : stor_elem_name
   INPUT : elem_name, field_name,...
   OUTPUT : elem_name,...
   BITS : <n:m>
   READ_TIME : <T_S, T_D>
   WRITE_TIME : <T_S, T_D>
```

NOTE:
Implied names for individual registers are:
stor_elem_name0 - stor_elem_name(n-1)

Figure 3-2. EXAMPLES OF STORAGE ELEMENT DESCRIPTIONS

```
MUX : mux_name
   INPUT : elem_name , field_name,...
   OUTPUT : elem_name
   CONTROL : field_name

PATH : path_name
   INPUT : elem_name , field_name,...
   OUTPUT :  elem_name
   CONTROL : field_name
```

Figure 3-3. EXAMPLES OF LINK ELEMENT DESCRIPTIONS

```
ALU : tran_elem_name
   INPUT : stor_elem_name, field_name, ...
   OUTPUT : stor_elem_name, ...
   BITS : ⟨ n:m⟩
   FUNCTION:
      value_name :
         transformation : ⟨ operation ⟩
         timing : ⟨ T_S, T_D ⟩
      .
      .
      .

SHIFTER : tran_elem_name
   [Same format description as ALU]

PSEUDO_TRAN_ELEM : tran_elem_name
   INPUT : stor_elem_name, field_name, ...
   OUTPUT : stor_elem_name
   BITS : ⟨ n:m⟩
   FUNCTION :
      transformation : ⟨ operation⟩
      timing : ⟨ T_S, T_D⟩

INTERFACE : tran_elem_name
   DATA_INPUT : stor_elem_name, field_name, ...
   CONTROL_INPUT : field_name, ...
   DATA_OUTPUT : stor_elem_name
   BITS : ⟨ n:m⟩
   FUNCTIONS :
      value_name :
         transformation : ⟨ operation ⟩
         timing : ⟨ T_S, T_D ⟩
      .
      .
      .
```

Figure 3-4. EXAMPLES OF TRANSFORMATION ELEMENT DESCRIPTIONS

```
⟨operation⟩ ::= ⟨basic operation⟩ | ⟨conditional operation⟩

⟨conditional operation⟩ ::= IF ⟨relational expression⟩
                            THEN ⟨basic operation⟩
                            ELSE ⟨operation⟩

⟨basic operation⟩ ::= ⟨extended variable⟩ = ⟨expression⟩

⟨extended variable⟩ ::= ⟨variable⟩ | ⟨variable⟩ ⟨field selector⟩

⟨expression⟩ ::= ⟨logical factor⟩
               |⟨expression⟩ .OR. ⟨logical factor⟩
               |⟨expression⟩ .XOR. ⟨logical factor⟩
               |⟨expression⟩ .NOR. ⟨logical factor⟩

⟨logical factor⟩ ::= ⟨logical secondary⟩
                   |⟨logical factor⟩ .AND. ⟨logical secondary⟩
                   |⟨logical factor⟩ .NAND. ⟨logical secondary⟩

⟨logical secondary⟩ ::= ⟨logical primary⟩
                      | .NOT. ⟨logical primary⟩

⟨logical primary⟩ ::= ⟨arithmetic expression⟩

⟨arithmetic expression⟩ ::= ⟨basic arithmetic expression⟩
   |⟨arithmetic expression⟩ [⟨condition code primary⟩]

⟨basic arithmetic expression⟩ ::= ⟨term⟩
   |⟨basic arithmetic expression⟩ + ⟨term⟩
   |⟨basic arithmetic expression⟩ - ⟨term⟩
   |-⟨term⟩

⟨term⟩ ::= ⟨factor⟩
   |⟨term⟩ * ⟨factor⟩
   |⟨term⟩ / ⟨factor⟩

⟨factor⟩ ::= ⟨primary⟩
           |⟨factor⟩ @ ⟨shift expression⟩

⟨shift expression⟩ ::= ⟨primary⟩
                     |⟨primary⟩ : ⟨primary⟩
```

Figure 3-5. SYNTAX FOR TRANSFORMATION ELEMENT OPERATIONS (1 of 2)

```
⟨primary⟩ ::= ⟨element⟩
            | ⟨element⟩  ⟨field selector⟩

⟨element⟩ ::= ⟨constant⟩
            | ⟨variable⟩
            | (⟨expression⟩)

⟨field selector⟩ ::= ⟨⟨expression⟩⟩
                   | ⟨⟨expression⟩ , ⟨expression⟩⟩

⟨relational expression⟩ ::= (⟨relational primary⟩)

⟨relational primary⟩ ::= ⟨expression⟩ ⟨relation⟩ ⟨expression⟩
                       | ⟨condition code expression⟩

⟨relation⟩ ::= .LT. | .LE. | .EQ. | .NE. | .GT. | .GE.

⟨condition code expression⟩::= ⟨condition code primary⟩
    | .NOT. ⟨condition code primary⟩
    | ⟨condition code primary⟩ ⟨condition code relation⟩ ⟨condition code
                                                          primary⟩

⟨condition code primary⟩ ::= ⟨discrete_name⟩

⟨condition code relation⟩ ::=.AND. | .OR. | .ANDNOT.

⟨variable⟩ ::= ⟨stor_elem_name⟩
             | ⟨field_name⟩

⟨constant⟩ ::= ⟨number⟩ | ⟨sign⟩ ⟨number⟩

⟨number⟩ ::= ⟨digit⟩ { ⟨digit⟩ }

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨sign⟩ ::= + | -
```

Figure 3-5. SYNTAX FOR TRANSFORMATION ELEMENT OPERATIONS (2 of 2)

3.1.3.1 Semantics for Transformation Element Operations

For each function specified in the Transformation Element description, the operation it performs must be specified by a statement which expresses the output as some function of the inputs. This function is expressed according to the syntax given in Figure 3-1, which provides sufficient expressive power to state arithmetic, logical, shift/rotate, bit, and conditional operations.

Each operation is specified as an assignment statement or a conditional. The condition may involve arithmetic comparisons or the discretes defined for the machine in the element description. The assignment statement specifies the output (or certain bits of the output) as an expression which may be logical, arithmetic, or shift.

Logical expressions use the relations OR, XOR, NOR, AND, NAND, and NOT.

Arithmetic expressions use the arithmetic operations of +, -, *, /, unary -. If a discrete is set as a result of the function, it must be specified in square brackets following the expression. For example, if ALUA and ALUB are the ALU inputs and ALUR is the output, and CRY is defined as a discrete element representing the carry, then

ALUR = ALUA + ALUB [CRY]

defines the operation of addition with the carry being set. Field selections may also be used to specify a portion of the whole data element width to which the function applies. A field specification is placed in angle brackets directly following the term to which it applies. Bit numbering uses the conventions specified the same as the ⟨n:m⟩ specifications in field and element descriptions. A field specification of

⟨n:m⟩

indicates that bits n to m inclusive are selected. The form

⟨n⟩

is a shorthand notation for the single bit ⟨n:n⟩.

Shift expressions include the specification not only of the number of bits to shift, but the input value shifted in as well. This is done using the general form:

```
BASE @ CNT : SI
```

where BASE is the item to be shifted, CNT is the number of bits to shift (positive count = left shift; negative count = right shift), and SI is the shift input into the shifted bit positions. For example, the operation for a simple 1 bit shifter attached to the output of a 16 bit ALU (bits 15:0), might be:

```
SHL : SHIFT_OUT = ALUOUT @ 1:0
SHR : SHIFT_OUT = ALUOUT @ -1 : ALUOUT <15>
```

The SHL would shift left one bit and input 0. The SHR would shift right one bit with sign extension. Rotates are handled by specifying the bits shifted out as the SI field; for example, a rotate right by 2 bits would be

```
RR2 : SHIFT_OUT = ALUOUT @ -2 : ALUOUT <1:0>
```

### 3.1.4 Microoperation Usage Rules

The usage rules of the microarchitecture description methodology give the set of rules for constructing valid microoperations. Although the different types of MOPs may require widely varying sets of resources, a general form of definition is possible. This general form is:

$$OP, \{I\}, \{E\}, \{O\}, \{T\}, \{F\}$$

where

| | |
|---|---|
| OP | is a mneumonic specifying the function to be performed |
| $\{I\}$ | is the set of possible inputs to be used as operands for the function |
| $\{E\}$ | is the set of transformational elements to be used to perform the function |
| $\{O\}$ | is the set of possible outputs (destinations) to be used for the results of the function |
| $\{T\}$ | is the set of timing increments used during the performance of the function |
| $\{F\}$ | is the set of control word bits used in specifying the function to be performed |

The usage rules are intended to specify the most primitive operations possible on the microengine, for example, the ALU operation rather than an ADD operation; or a memory read rather than loading a memory location into a register. The set of these primitive MOPs forms a third intermediate form in the overall compilation process, called IL3. There are a relatively small number of types of MOPs at this low level into which all higher level operations, in particular IL2 operations, can be decomposed. The decomposition process is termed instruction set interpretation (ISI) and is addressed in Section 3.2.

The usage rules are derived from the microinstruction control capability and the transformational elements' function capability. The I and O sets typically represent a storage element (or pseudo-element). Note that for some operations the I or O set may be empty (denoted by $\emptyset$). For example, the MOP that sets/resets the interrupt enable has an empty I set. The T set represents the time increments in which the E set is active. Time increments are

arbitrarily defined to be 1 nanosecond per increment. The T set for a MOP thus represents the contiguous time intervals, within the cycle, during which the {I}, {E}, and {O} resources are active and unavailable for other MOP usage. For those MOPs whose execution results in multi-cycle activation times for various resources, a syntactical form is defined. Note that for multi-cycle activated elements such as large data memories, external floating point units, and the like, a pseudo-element is defined which represents the functional interface to the element. This technique allows for the separation of the resource into an activation time within the cycle (for the interface) as well as an activation time for the actual resource. This capability will allow for more efficient utilization of resources which have a 'look-ahead' capability, a pipelined mode of execution, etc.

The types of MOP usage rules (Dasgupta's 'organizing mechanisms') required for instruction set interpretation are, at least, those to do the following functions:

1. ALU
2. Shift
3. Movement
4. Address Formation
5. Control
6. Interface
7. Data

To derive the basic operations needed in the MOPs, the usage rules are used in a procedure as follows. The IL2 instruction set is compared with the capabilities of the microinstruction format. The OP operations selected are only those that form the union of the sets:

> OP is an operation which exists in both the IL2 tuple and the microinstruction format

and

> OP is not an IL2 operation, but is a primitive operation of the basic type needed to decompose an IL2 operation.

With the OP operation determined, the I set and O set resources must be selected so that the execution of OP, {I}, {O} is a primitive operation of the machine. This means that {O} must be chosen

as the next immediate destination of OP, {I}. The coupling of information from the {I}, {O} sets of one MOP to another can be accomplished by a movement primitive when necessary.

As an example, consider the TRW 2AU-80 (B side only ) [Ref. 34, 35]. If we select registers AM and BM as the storage element inputs to the ALU, there are several choices for output storage elements. RAM, BM, BL, AM, AL and ALUM are all possible storage elements to receive the results of the operation. When we are obligated to select the storage element which is the next immediate destination, we select ALUM. The transfer of ALUM contents to other storage elements such as RAM, BM, or BL must be accomplished by the movement primitive.

This definition of machine primitive is necessary to achieve the flexibility needed in the code generation and improvement step of the compilation process.

The execution of a microinstruction is controlled by the fixed timing of a control store cycle. MOP timings are determined by the multiple minor or polyphase timing periods within the control store cycle. Examples of these minor cycles are time to load registers, time to settled ALU output, time until the RAM address registers are latched for a read or write operation or the time to perform an ALU operation. This latter timing period could involve multiple control store cycles (a multiply or divide operation for example).

The timing determinations for the MOPs are logical minor cycles and, thus, may include one or more physical minor cycles. This is particularly true in asynchronous machines where detailed timings on primitive operations are indeterminate in many instances.

The choice of fields for assignment to the MOP is obtained directly from the microinstruction description. For example, the 80 bits of the AU Command microinstruction format in the TRW 2AU-80 is divided into 31 fields. Each OP, {I}, {U} which performs an AU operation uses several fields.

The use of {E} set here refers to the physical transformation unit which is to perform the OP, {I} {O} MOP along with the link units needed to route information. The choice of this unit is made based upon the above discussions of primitive operation selection. However, this selection is made with a more global view in mind because of the conflict resolution which must take place between MOPs in accomplishing the execution of the entire IL2 statement.

The procedure is to decompose the IL2 statement into a sequence or sequences of MOPs which collectively accomplish execution. This sequence(s) is then assigned the transformation unit resources of one or more microinstructions (typically, minimum sequences of MOPs generate the least number of microinstructions.)

An example of each of the above seven types of MOPs is as follows:

1. ALU MOP usage rules are the general rules for using the ALU(s), rather than its specific functions. Elements involved are the ALU(s) and those storage elements (or pseudo-elements) which are the immediate inputs and outputs. An example of the expression of an ALU usage rule is:

   ADD, $\{R_i, R_2, \text{Carry=0}\}$, $\{\text{ALU1}\}$, $\{\text{R3, Carry}\}$, $\{T_3\text{-}T_{15}\}$, $\{F\}$

   where $R_i$ denotes a general purpose register described in the Element Description and {F} is derived from the Microinstruction Description.

2. SHIFT MOP rules define the use of the shifter(s), in terms of the elements (or pseudo-elements) which are immediate inputs and outputs. An example is:

   SRL, $\{\text{ALUOUT, LINK}\}$, $\{\text{shifter2}\}$, $\{R_3, \text{LINK}\}$, $\{T\}$, $\{F\}$

3. MOVEMENT rules are the rules for moving data between storage and/or transformation units. Almost any hardware component can be involved in a move. Note that no non-trival transforming operation is performed on the data. An example is:

$$\text{MOVE}, \{\text{XBUS}\}, \{\text{PATH5}\}, \{R_3\}, \{\text{T}\}, \{\text{F}\}$$

4. ADDRESS FORMATION rules specify the possible methods of forming the address of the next microinstruction to be executed. These rules are perhaps the most difficult to derive because of the wide diversity of addressing schemes available in microarchitectures. An example of this type of usage rule is:

$$\text{JUMP}, \{\text{CSAR, MI, CARRYSET}\}, \{\text{CSAP}\}, \{\text{CSAR}\}, \{\text{T}\}, \{\text{F}\}$$

where CSAR is the control store address register, MI is the address data from the microinstruction, carryset is a testable condition for carry set, and CSAP is the control store address processor.

5. CONTROL MOP rules are used to set/reset the discretes of the micro engine. An example, for setting the interrupt enable, is:

$$\text{SETIE}, \emptyset, \{\text{INTERRUPT PROCESSOR}\}, \{\text{LATCH3}\}, \{\text{T}\}, \{\text{F}\}$$

6. INTERFACE MOP rules provide the ability to interface with data memories, floating point units, input/output units, etc. Typically the elements involved are control and data registers. For purposes of this description procedure, the external unit itself is treated as a 'black box'. An example of a data memory operation is:

$$\text{READ}, \{\text{MAR1}\}, \{\text{MEMI1}\}, \{\text{MDR}\}, \{\text{T}\}, \{\text{F}\}$$

7. DATA usage rules are the means of introducing absolute numerical quantities for addressing purposes and for literals. Elements involved are constant memories or literal capability in the microinstruction itself. An example of a fetch of a constant from the constant memory is:

   LOADCON, {CAR}, {CMEMI}, {XBUS}, {T}, {F}

   where CAR is the constant memory address register.

The T set values in the above rules are derived from the combined timings specified for the individual elements involved (from the Element Descriptions); while the F set is the union of the fields associated with each of those elements.

### 3.1.5 Micro Engine Behavioral Rules

The micro engine behavioral rules specify the interactions among the MOPs. The purpose in specifying the behavior is to permit the detection of conflicts which would prevent MOPs from being executed concurrently (i.e., in the same microinstruction). These rules consider the inherent timing, resource, control word bit, and possible data dependency conflicts between pairs of MOPS.

To derive the benefits of horizontally microprogrammed processors, it is desirable to execute as many primitive operations (MOPs) as possible within the timing period of one control word cycle. To accomplish this, the detailed knowledge of the timing and resource utilization required by each MOP, as specified in the sections above, is necessary. With this information in hand, a determination of the parallelism of MOPs can be made.

Two MOPs are said to be parallel if the execution of the MOPs in the same control word cycle produces the same result as would be obtained by the execution of the MOPs in separate control word cycles. The detection of parallelism is generally divided into target microengine dependent rules and target microengine independent rules.

The target microengine dependent rules of behavior are determined by examination of the characteristics of the specific target microengine. As an example, consider targets in which there are several different microinstruction formats possible. Each format can typically be executed in one control word cycle. Thus, MOPs available with different formats can never be executed together (i.e., they are never parallel). As another example, in the TRW 2AU-80 [Ref. 34, 35] the multiply operation uses the BM and BL registers as source storage elements for the operands. Thus, the BM and BL registers cannot be loaded by another MOP until the multiply operation has finished. In this case, any MOP containing the BM or BL register in the I set or O set cannot be placed in the same control word cycle(s) as the multiply operation.

The rules for dependent behavior are incorporated into the microengine behavior rules in an 'ad hoc' manner. Their representation will preferrably be in matrix form very similar to those used below for the independent rules.

The target microengine independent rules of behavior are determined from the usage rule representation of the MOP. They include data dependency determinations that affect parallelism as well as the determination of conflicts among resources and timing. The specific resource conflicts of interest are: 1) conflicts among the storage elements of the I and O sets within a sequence of MOPs, 2) conflicts in the use of control word bits and fields, 3) conflicts that arise from timing overlaps, and 4) conflicts due to the use of E set units (transformation and path units). The rules for independent behavior are discussed in what follows.

Let $MOP_i$ and $MOP_j$ be two MOPs from the usage definition section. Then we have:

$$MOP_i : OP_i, \{I_i\}, \{E_i\}, \{O_i\}, \{T_i\}, \{F_i\}$$

$$MOP_j : OP_j, \{I_j\}, \{E_j\}, \{O_j\}, \{T_j\}, \{F_j\}$$

Data interactions can be summarized by the following situations:

- If $MOP_i$ precedes $MOP_j$ and $\{O_i\} \cap \{I_j\} \neq \emptyset$
- If $MOP_i$ precedes $MOP_j$ and $\{O_i\} \cap \{O_j\} \neq \emptyset$
- If $MOP_i$ precedes $MOP_j$ and $\{I_i\} \cap \{O_j\} \neq \emptyset$

If any of these situations occurs there is a data dependency between $MOP_i$ and $MOP_j$ and a change in their order of execution would most likely produce a result different than execpted. One must also be concerned with interactions which are more subtly encountered. Suffice it to say that $MOP_i$ and $MOP_j$ may be separated by many other MOPs, each of which may not directly violate the above rules, but will none-the-less chain the data dependence of $MOP_i$ and $MOP_j$.

The data interactions amongst MOPs are used during the code generation and improvement step of the compilation process to validate execution concurrency capability between pairs of MOPs.

Conceptually, for non-data dependent conflicts, the behavior description consists of a square matrix, B, whose rows and columns represent the MOPs defined in the usage section. The entry at the position Bij is 1 if a conflict exists between MOPs i and j, and 0 if no conflict exists. This is logically equivalent to a list of pairs of MOPs for which a conflict does exist with regard to elements, fields, or timing. To derive the behavior matrix, the possible causes of conflict must be investigated separately and then combined.

Let MOPi and MOPj be defined as above. We then define three intermediate matrices, as follows:

1. The resource conflict matrix, $R_{ij}$, is defined as:

$$R_{ij} = \begin{cases} 1 & \text{if} \left\{I_i\right\} \cap \left\{I_j\right\} \cap \left\{O_i\right\} \cap \left\{O_j\right\} \cap \left\{E_i\right\} \cap \left\{E_j\right\} \neq \emptyset \\ 0 & \text{Otherwise} \end{cases}$$

   Thus, $R_{ij} = 1$ imples that some conflict exists between the set of input and output elements of the MOPs.

2. The field conflict matrix, $F_{ij}$, is defined as:

$$F_{ij} = \begin{cases} 1 & \text{if } \left\{F_i\right\} \cap \left\{F_j\right\} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

   Thus $F_{ij} = 1$ implies there is a conflict for control word bit usage between the MOPs.

3. The timing conflict matrix, $T_{ij}$, is defined as:

$$T_{ij} = \begin{cases} 1 & \text{if } \left\{T_i\right\} \cap \left\{T_j\right\} \neq \emptyset \text{ and } R_{ij} = 1 \\ 0 & \text{otherwise} \end{cases}$$

   Thus, if $T_{ij} = 1$, some overlap exists in the timing of the MOPs. However, this must be further refined. It is clear there is a conflict only when the same resource(s) is involved. Hence, one must now examine the $R_{ij}$ matrix.

Given these three sets, we can then form the matrix, $B_{ij}$, for non-data dependent conflicts according to the following rules:

- If $F_{ij} = 1$, then $B_{ij} = 1$
- If $F_{ij} = 0$, and $T_{ij} = 0$, then $B_{ij} = 0$
- If $F_{ij} = 0$, and $T_{ij} = 1$, then $B_{ij} = R_{ij}$

Thus, $B_{ij}$ matrix represents the interactions amongst MOPs due to field, timing and/or resource contention. This matrix is used during the code generation and improvement phase of the compilation process to verify the ability to combine MOPs in forming a microinstruction, i.e., the ability of MOPs to be executed concurrently.

## 3.2 INSTRUCTION SET INTERPRETATION

Using the procedures developed in Section 3.1 above to collect and describe the necessary target microengine information (perhaps requiring several iterations through the procedures), the next step in the process is to use this information to interpret and decompose the more functionally complex IL2 statements. Thus, the preceding sections have shown not only how the control word and hardware components of a micro engine may be described, but how valid MOPs may be formed and legally combined. We can think of these sections as specifying the syntax, or the form, of the machine. In order to create microprograms, we must also know the semantics--the meaning--of the microoperations. This will be used to effect the decomposition of IL2 statements.

The IL2 language is designed with the following goals in mind:

- To be machine independent
- To correspond as nearly as possible in a one-for-one manner with MOPs generally available in typical target microarchitectures, but without 'losing' any information which may permit more efficient microcode generation
- To include those statements and capabilities necessary for decomposing each IL1 statement as efficiently as possible.

The format of the language is

Label, OP, I, O

where

Label is a statement label
OP is the operation code
I is the source operand
O is the destination operand

The set of primitive IL2 statements, in this format, selected are those which are necessary and sufficient for efficient decomposition and execution of IL1 statements.

Even though the IL2 statements specify very basic operations in a generalized form, it still may be necessary to expand their operation into a series of MOPs on the specific target micro-architecture. Moreover, the statements are not yet bound to any actual physical elements on the target microengine. This expansion, resource allocation, and binding process is what occurs in the ISI, code improvement and code generation phases of the process. In particular, ISI focuses on the techniques and procedures used to implement those IL2 statements which expand into one or more sequences of MOPs. Resource allocation and binding considerations significantly add to the complexity of ISI.

The interpretation of IL2 statements can be accomplished in two ways. The first, referred to as static interpretation, is achieved by selecting, in a static manner, a sequence of MOPs which will accomplish the actions embodied in the IL2 statement specification. The second, referred to as dynamic interpretation, is achieved by selecting a sequence of MOPs in light of currently available machine resources. The impact on the resulting code generated, using only static interpretation, is usually adversely significant. Thus, interpretation is most often accomplished by a combination of these techniques.

The decomposition sequence is termed a template, and becomes, for microcode generation, similar to a macro expansion for IL2 opcodes into a sequence of MOPs. At least one template must be provided for each IL2 instruction. It is not altogether clear how to permit the templates to reflect the dynamic state of the microengine. It may be sufficient, however, to specify a number of static templates to permit the code generator to select the one which best matches the current state of the microengine resources. The issue is closely bound to the general resource allocation scheme used by the code generator.

Thus, an overview of the ISI procedure might be the following:

1. Define a macro form expansion for each IL2 statement; call this the standard template.

2. Determine methods and techniques for assessing the resource requirements and timing information of interest for these standard templates.

3. Using the standard template, devise a means of indicating alternate, but equivalent, ways to accomplish the function/operation (equivalent both in terms of resulting action and in terms of resource capability).

4. Devise a means to modify the standard template, at time of use, to accommodate currently available microengine resources.

5. Determine a means of assessing penalties associated with these modifications so that a 'best fit' can be chosen.

The above procedure would be significantly simplified, from a practical viewpoint, if the usage rules (IL3 statements) given earlier could be used to derive a means of assessing the equivalency of MOPs or MOP sequences in terms of function (operation), resource requirements and utilization, and timing information. (Although only start-to-stop timing for the MOP sequence is directly of interest, the timing associated with indirectly affected elements is of major concern). It is apparent from efforts to date that the MDM can be used to validate MOPs formation and thus, to form valid sequences to be used to establish a standard template. However, resource equivalency and interchangeability, as well as standard template selection and formation, are processes which require judgment and thus a high level of artifical intelligence.

Another scheme which appears to have promise is the following. To start with, instead of building a template for each IL2 statement type, a MOP tree is built with branches denoting different, but appropriate, choices for the selection of each MOP in the execution sequence, and an associated MOP busy or free resource utilization table is built. Thus, as each IL2 statement is interpreted , a MOP tree is traversed generating a MOP execution sequence using only available (free) MOPs.

This means of interpretation has significant advantages in terms of efficiency of code generated; however, depending on the complexity of the IL2 statements, MOP trees may be difficult to define. Additionally, compilation times are longer and, for very 'branchy' trees, may be prohibitive. Suffice it to say that MOP trees must be judiciously chosen and limited in extent.

## 3.3 PRELIMINARY EXPERIMENTATION

To aid in the derivation and development of the techniques and concepts under study, test microarchitectures were constructed, and drawn upon from industry, to highlight the specific areas of interest. Additionally, to reduce the complexity of analyzing the architectures, simplifying initial conditions, assumptions, and constraints were established to provide a test bed. Since the study schedule did not allow for any extensive experimentation, the test architectures were used to concentrate efforts on the more critical steps in the Microarchitecture Description Methodology.

A rather general purpose bus oriented microarchitecture was developed and refined for use in testing description concepts and techniques. The model has a register oriented ALU, but incorporates a capability for several levels of memory (scratch pad, fast RAM, slower main memory, etc). The model also allows for external (special) device interfacing through the bus structure. The microengine itself has a shift capability, significant discrete control flexibility, and significant test condition and microaddressing capability. This model serves as the 'strawman' architecture for description techniques and methodologies. A simplied block diagram of this microarchitecture is given below in Figure 3-6. This architecture is typified by the microengines of references [23], [14], [1,25], [36,37], [29], and [21,22].

An example of a typical simplified microinstruction for the 'strawman' class of microarchitectures is given in Figure 3-7. Other examples can be found in [25], [36,37], and [14]. In particular, [14] presents the microinstruction layout for a stack oriented architecture.

A second type of microarchitecture was also used in testing the MDM concepts developed. This is an asynchronous architecture typified by the microengine of reference [35]. These architectures are also called DATA FLOW architectures because they are data or event driven and are of particular interest in signal processing applications. A simplified block diagram of this type of microarchitecture is presented in Figure 3-7. An example of the microinstruction layout and contents for this type of microengine can be found in reference [35].

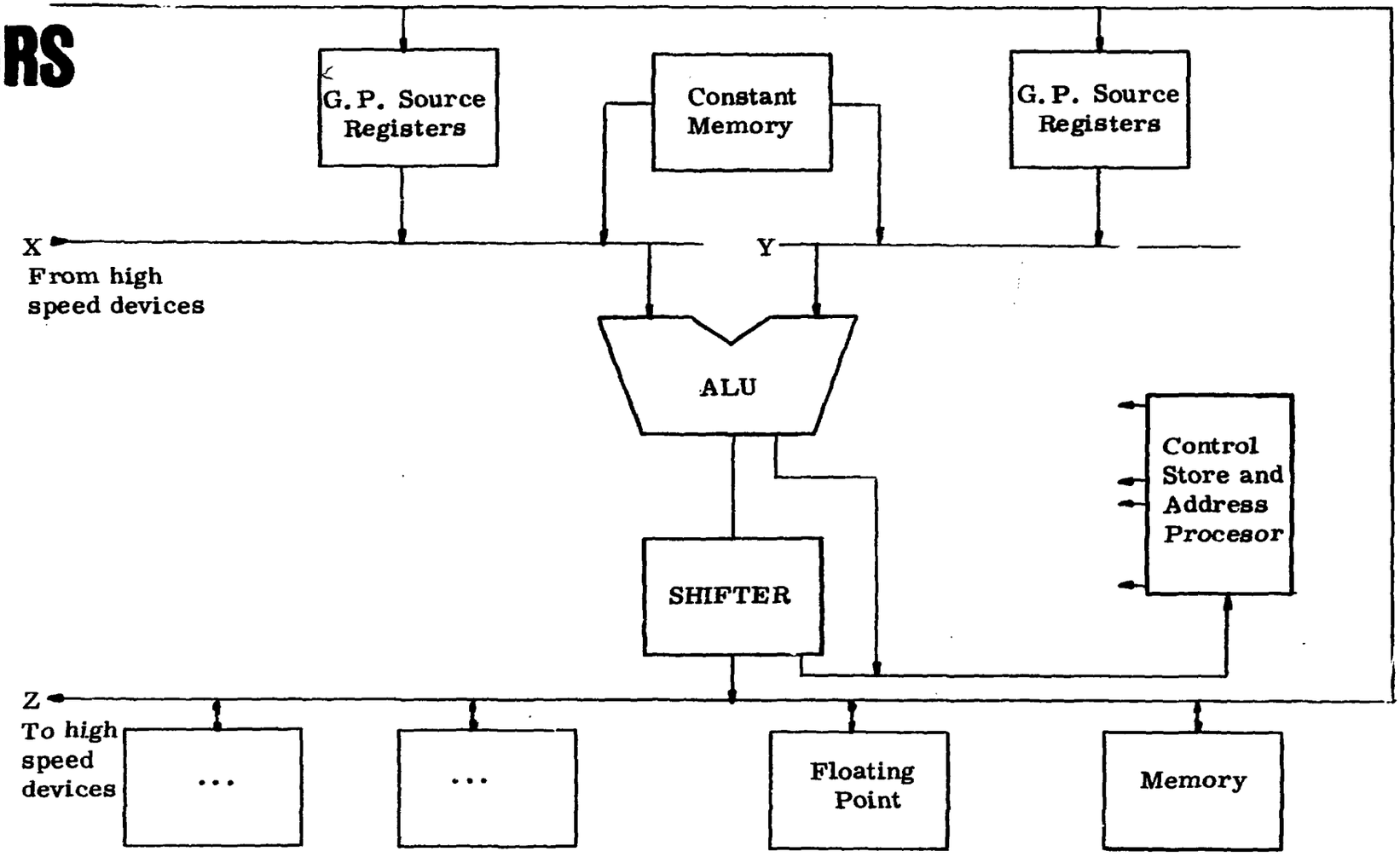



Figure 3-6. MDM 'STRAWMAN' MICROARCHITECTURE BLOCK DIAGRAM

| Storage and ALU Control | | | | | Discrete Control | Peripheral Device Control | Execution Control | | |
|---|---|---|---|---|---|---|---|---|---|
| Source X | ALU | Source Y | Shifter | Destination Z | Discrete Conditions | External, Offline, Memory Control | Condition Codes | Test Conditions Selector | Address Control |



Figure 3-7. MDM 'STRAWMAN' MICROARCHITECTURE MICROINSTRUCTION EXAMPLE

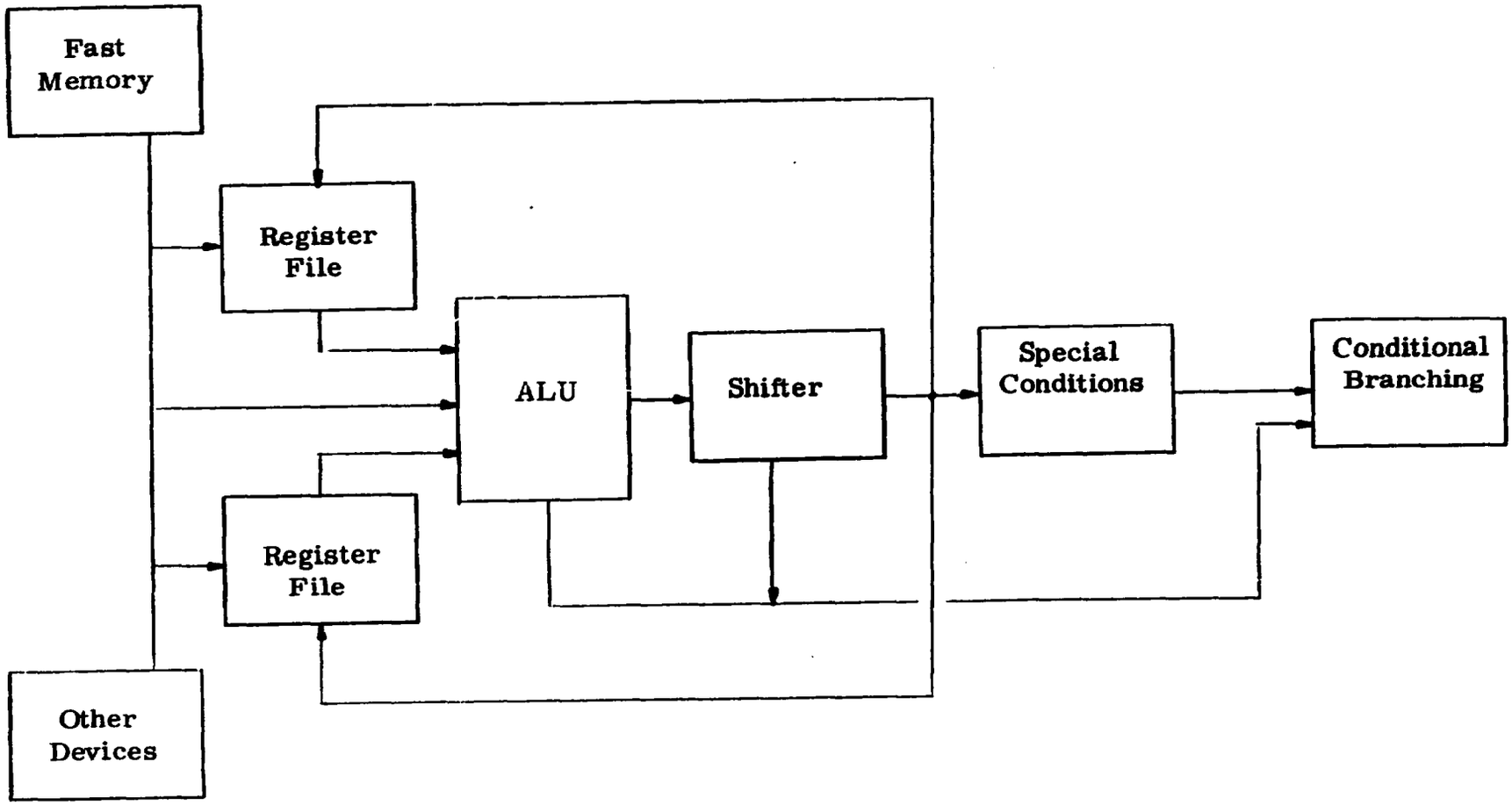



Figure 3-8. DATA FLOW MICROARCHITECTURE BLOCK DIAGRAM

### 3.3.1 Specific Target Microarchitecture Assessments

Several specific target microarchitectures were chosen to be used to ascertain the 'fit' of the MDM. The architectures chosen represent a wide diversity of commercially developed processors. Also included are processors based on popular bit-slice microprocessors. The MDM evaluation and assessment of the latter is very difficult because of dependencies on implementation preferences and philosophies not available to the author at this writing. The targets chosen are the following:

- AMD 2901 Series
- DEC VAX 11/780 Processor
- TRW 2AU-80 Processor

#### 3.3.1.1 AM2901 Based Processors

The processor of reference [ 25 ] is one of several using the AM2901 series devices. (Similarly, there are processors based on the MC68000 and the MC10800 series devices. The AM2901 series was chosen because of the authors familiarity with this series). The AM2901 series devices are four-bit microprocessor slices consisting of a 16-word by 4-bit two-port RAM, a high-speed ALU, and the associated shifting, decoding and multiplexing circuitry. The 9-bit microinstruction word is organized into three groups of 3-bits each and selects the ALU source operands, the ALU function, and the ALU destination register. The microprocessor is cascadable with full look-ahead or ripple carry, has 3-state outputs, and provides various status flag outputs from the ALU. For more details, the reader is referred to reference [25].

The MDM description of processors that are AM2901 based pose no severe problems, in general. There are some trouble spots, however. In particular, those implementations that use a multiply/divide step approach to multi-precision arithmetic tend to be difficult to describe in terms of timing because of the complex inter-relationships amongst participating elements. Notable among these elements are the double shift elements, i.e., the shift extension register, and the shift control multiplexors.

### 3.3.1.2 DEC VAX 11/780 Processor

The VAX 11/780 is a multiprogramming computer system. The system uses a 32-bit architecture, memory management and a virtual memory operating system to provide essentially unlimited program address space. The processor has a variable length instruction set and a variety of data types including decimal and character string types. The central processing unit of the VAX 11/780 includes:

- 8 byte prefetch instruction buffer
- T ime of year clock.
- Programmable real time clock
- Floating point accelerator
- Writable Control Store (12k bytes)
- Writable Diagnostic Control Store (12kbytes)
- 128 entry address translation buffer
- 8k byte cache memory (two-way associative)
- Integral memory management
- Sixteen 32-bit general registers
- 32 interrupt priority levels
- Intelligent console interface

The micro-engine is composed of 8 separate asynchronous sections:

- Arithmetic
- Exponent
- Address
- Data
- ID Bus
- General control
- Micro-sequencer
- Interrupt

The VAX 11/780 Writable Control Store Option consists of 1024 96-bit words occupying addresses 1400 through 17FF. Each word also contains 3 parity bits. The control store word is separated into 3 banks for loading and executing purposes:

Bank 0: bits 0-31 (numbering right to left)
Bank 1: bits 32-63 (numbering right to left)
Bank 2: bits 64-95 (numbering right to left)

Many of the micro-fields are multipurpose so that the control word has numerous formats. The basic microcycle is 320ns; however, certain activities can cause a microinstruction to exceed this.

The documentation available, covering the microengine architecture and microprogramming of this processor, is given in references [36, 37]. Although this documentation is well written and somewhat detailed it is less than adequate to allow one to use and evaluate this processor for microprogramming purposes. In particular, little information is given about microoperation relationships and no explicit information is given about intra-cycle timing and timing relationships in general.

This microengine is relatively easy to accommodate with the MDM in terms of the microinstruction description and the element description. However, due to the very close inter-relationships with the resident standard microinstruction decoding microcode, the usage rules (IL3 statements) are somewhat difficult to derive and the behavioral rules are many and complex. The behavioral rules tend to degrade into 'ad hoc' and exception rules because of the design orientation toward hardware which executes a macroinstruction set most efficiently. Of particular difficulty are the usage and behavioral rules associated with hardware-assisted features used for emulation of the macro-instruction set. Four additional areas of difficulty are:

1. The cumbersome and complex main memory interface (the cache in particular);
2. The use of a constant memory which involves a multi-cycle operation;
3. The restrictions on general purpose register usage (due to their dedication to macro-instruction functions); and
4. The cumbersome and complex interface among asynchronous sections (for example, the Exponent Section).

### 3.3.1.3 TRW 2AU-80 Processor

The TRW 2AU-80 processor is described in detail in reference [35]. As mentioned earlier, this processor represents an asynchronous architecture, designed primarily for signal processing applications. In an asynchronous architecture, operations are

caused by events as opposed to clock timings. In many instances, operations are triggered by any one of a combination of inputs being changed. Thus, if one were to determine the conditions for use of a specific element at a specific point in time in the microcode, one could require significant 'a priori' knowledge of the previous usage of related elements; not only from the earlier stages of the current microcycle, but perhaps from actions specified in several previous microcycles. This inability to exactly specify primitive operation timings makes a general machine description very difficult. A synchronous machine on the other hand has a very well-defined timing relationship amongst its primitive operations. One has only to be concerned with the actions taken at the point in time of interest.

Since the TRW 2AU-80 is an asynchronous architecture, initial attempts at microarchitecture description techniques took the form of a resource and timing model specification (See reference[12]). The reason for this is as follows. In order to improve microcode efficiency by local compaction, one must be able to determine the state of the targets' resources at various decision points. (It is also clear that any global optimization algorithm relies heavily on this determination as well.)

Thus, in applying this to the TRW 2AU-80, the non-deterministic nature of the processor leads to an inability to exactly specify primitive microoperation timings. Thus, for example, instead of doing a register load operation by building a route, using primitives like multiplexors and path designators, in a one-after-the-other sequence based on knowing specific timing relationships, a complete route for the desired register load would have to be denoted along with its complete and exact timing. Even through the use of this broader definition, the 'cost' in time of using a specific route, at a specific point in the microcode, can not always be calculated with sufficient accuracy to assure reliable code. Moreover, the impact on the efficiency of the code generated and on the compilation time required due to the above definition is difficult to assess.

Particularly troublesome areas for the MDM are the following:

- Path/Function element descriptions
- Ascertaining the time associated with the use of only the path/function element

- Ascertaining the time associated with the use of the path/function element and the relationship with 'a priori' conditions
- Determing the conflicts in other resources associated with the use of a particular path/function

Another architectural feature of the target processor which impacts the description process is its ability to parallel process. Although the TRW 2AU-80 has two arithmetic units, they are both serviced by a single control store and control store sequencer and thus, are operationally very dependent. Moreover, the two arithmetic units, without an interrupt or communication/control link, pose significant difficulties for 'true' parallel processing because there is no describable control mechanism to allow for synchronizing the processing. With the arithmetic units sharing some working registers in performing certain arithmetic operations, a 'time-out' situation is the only means of precluding conflicts between the units. This lack of describable control minimizes the parallelism of a horizontal processor in an automatic microcode generation system.

## 4.0 CONCLUSIONS AND RECOMMENDATIONS

The conclusions and recommendations which follow are an attempt to address the more substantial issues and concepts which play a dominant role in the microarchitecture description methodology process. Because of the wide variation of some of the more significant parameters involved in microarchitecture description and the limited amount of study time, feasibility was approached with 'soft' criteria as opposed to the more formal 'pass-fail' criteria for evaluating the techniques and concepts developed.

### 4.1 Microarchitecture Description Conclusions

Based on limited experimentation and application, the microarchitecture description methodology described earlier in this document appears to be both feasible and practical for use in the HLL-to-microcode compilation process to describe and interject the target processor characteristics. The parameters derived and developed in establising the methodology are an attempt at developing an invariant microengine description procedure. These parameters are very idealistic, in that they represent the parameters (and their form and information content) in the way that is most desired. However, this approach lends itself to quantification most easily and allows for establishing the basic 'ground' rules for description purposes. Using these rules, ad hoc interpretations can be made.

The microinstruction description of the microengine gives an indication of the kinds of control available as well as the flexibility available in exercising this control. The description attempts to provide syntatical and semantical capability to allow the latter 'usage rule' section to use these efficiently. Similarly, this descriptive capability is applied to the data manipulation and computational functions available in transformation units, the resource characteristics and allocation/deallocation complexity of storage elements and the flexible in data flow control available in the path or link units in the element description section.

Although the syntax and semantics for the microengine timings of interest have been provided for, it is felt that much more detailed experimentation is necessary to 'flesh-out' this area of the MDM. The timing parameterization of the microengine has the most

dramatic impact on microcode density and thus, on microcode efficiency. The importance of the ability to determine timing in sufficient detail cannot be overstressed when microcode efficiency is of concern. Of particular interest is the capability for more detailed description of multi-cycle MOPs. These MOPs tend to be highly inter-related with the other elements and so tend to cause usage constraints and highly complex usage rules. Similarly, more practical experimentation with the behavioral features of microarchitectures is needed to capitalize on the concurrency available in MOP formation. This again has a very significant effect on the generated microcode efficiency.

Finally, the control store next address formation issue is of concern. Although provisions in the MDM allow for the description and usage of an 'organizing mechanism' labeled the Control Store Address Processor, the wide variation amc gst microengine implementations of this capability precludes relatively easy and simple description. Moreover, this mechanism is highly inter-related with the microinstruction description and the action and timing of the remaining microengine elements. Since this also affects microcode efficiency, the MDM must be extended, through experimentation, to allow for very detailed syntatical and semantical descriptions of this area of the microarchitecture.

## 4.2 Instruction Set Interpretation Conclusions

As was mentioned in Section 3.2, the ISI issue is closely bound to the general resource allocation scheme used by the code generator. There is virtually no limit to the kinds of resource allocation/deallocation schemes which may be employed in Instruction Set Interpretation. The crucial question is: In terms of code generation overhead costs, at what point does the worth of the schemes lead to little or no value in terms of execution efficiency?

It is clear from previous work that, particularly for microengines with parallel processing capability, a significant improvement in execution efficiency can be obtained by the use of appropriate allocation/deallocation schemes. The value of a particular scheme is directly related to target machine microarchitectural considerations. This is especially the case with storage element allocation/deallocation schemes in machines with several hierarchical levels of memory.

The most important elements to be allocated are the microengine transformation unit (or units) and its storage elements (especially registers). Transformation units may be allocated by maintaining a utilization table with 'free/busy' status and timing information. Storage units may be allocated in many ways, such as FIFO, frequency-of-use, etc. The approach favored is to compute a 'penalty' associated with deallocation of each variable, based on usage, variable size and type, and allocate new variables during ISI, as required, to those resources for which the deallocation penalty is smallest.

From a compilation process point of view, it appears that the most severe disadvantage and/or limitation of resource allocation/deallocation schemes is due to the generation and computations inderent in using tree structures and tree structure decision making during the ISI phase.

Besides the general resource allocation/deallocation schemes used, the ability to use the usage rules derived in the MDM to dynamically form MOP templates for IL2 decomposition is of significant value in the ISI process. The ability to form equivalent MOP templates based on currently available resource situations would allow for highly dense and efficient microcode. If this process can be automated to dynamically interact in the microcode generation process, the practicality of the MDM will be significantly enhanced. Moreover, the ease of and time for retargeting the HLL-to-microcode compilation process will be greatly simplified and shortened.

## 4.3 Recommendations

The MDM concepts presented earlier in this document form the basis for a procedure which will simplify and speed-up the retargeting of the HLL-to-microcode compilation process discussed in Section 2.0. However, continued research and experimentation is needed which is directed towards establishing the practicality and efficacy of the MDM in the compilation process (see Appendix B).

The need is for more indepth and thorough experimentation and evaluation across a wide spectrum of microarchitectures. In particular, parallel and pipelined archiectures should be investigated and experimented with as well as conventional architectures and the more contemporary data flow architectures. The continued research and experimentation in diverse microarchitectures should focus on 'fleshing-out' the MDM in microengine detail description and in providing a more general and complete syntatical and semantical basis for ISI. Additional areas requiring further efforts are:

- the development and use of dynamic ISI techniques
- the development and use of resource allocation/ deallocation techniques to aid in dynamic ISI
- the development and use of a more generalized microcode generator to aid in experimentation in the above areas

An additional objective of continued research and experimentation is an assessment of the feasibility of using the MDM to produce verification and validation tools to be applied to the microcode generated by the compilation process, as enhanced by the above. Given a description of a target processor, using the MDM discussed above, and given a means for assimilating this description to do the ISI step with appropriate resource allocation schemes and capabilities, the next significant problem one encounters is a means for verifying and/or validating the microcode generated by the compilation process. Work done to date strongly recommends the use of the MDM to provide some solution to this problem.

It appears likely that the MDM can be enhanced to provide for the automatic generation of a Simulator for the described target machine. This would allow for the symbolic execution of the generated microcode to verify correct microinstruction formation as well as establishing the validity of the resulting microprogram.

This capability would be of significant benefit to the compilation process and to further research and experimentation in the pieces of the process. Of particular significance is the capability to perform very detailed evaluations of microarchiectures, in relationship to specific processing algorithms, in a quick, direct, and measurable way.

JRS

With the development of tools and techniques in the above mentioned areas and continued study and experimentation of the entire process, strong preliminary indications are that the benefits of an easily retargeted automatic microcode generation system will be realized.

## 5.0 BIBLIOGRAPHY

[1] Advanced Micro Devices Data Book (AMD2900 data), Advanced Micro Devices Corporation, Sunnyvale, Ca., 1976.

[2] Advanced SMITE Technical Report (Interim): Performance Measurement, Extensibility, and Concurrency Study, TRW Corporation, Defense and Space Systems Group, March 1978.

[3] Agrawala, A.K. and Rauscher, T.G., Foundation of Microprogramming Architecture, Software, and Application, Academic Press, Inc., 1976.

[4] Baba, T. and Hagiwara, H., "The MPG System: A Machine-Independent Efficient Microprogram Generator", IEEE Transaction on Computers, Vol. C-30, No. 6, June, 1981, pp. 373-395.

[5] Barbacci, M.R., Barnes, G.E., Cottell, R.G., and Siewiorek, D.P., The ISPS Computer Description Language, Carnegie-Mellon University, Pittsburgh, Pa., March 1978.

[6] Blou, J.S., Holland, C.J. and Keating, D.L., "The Micro-Architecture of the Eclipse MV/8000-Conceptions and Implementation," Proceedings of the 13th Annual Workshop on Microprogramming, November 1980.

[7] Bondi, J.O. and Stigall, P.D., "HMO, A Hardware Micro-code Optimizer," Proceedings of the Second Annual Symposium on Computer Architecture, January 1975.

[8] Chu, Y., "Introducing CDL," Computer, Vol. 7, No. 12, December 1972, pp. 31-33.

[9] Dasgupta, S., "Some Aspects of High-Level Microprogramming", ACM Computing Surveys, Vol. 12, No. 3, September 1980, pp. 295-323.

[10] DeWitt, D., "A Control Word Model for Detecting Conflicts Between Micro-Operations," Proceedings of the Eighth Annual Workshop on Microprogramming, October 1975.

[11] Drongowski, P.J. and Rose, C.W., "Application of Hardware Description Languages to Microprogramming: Method, Practice, and Limitations," Proceedings of the 12th Annual Workshop on Microprogramming, November 1979.

[12] Final Report for the Study of Compilers for High Throughput Horizontal Microprocessors, JRS Industries, Inc., San Diego, Ca., November 1979.

[13] Final Technical Report for Higher Order Language for High Throughput Horizontal Microprocessors, JRS Industries, Inc., San Diego, Ca., October 1980.

[14] GPH Processor Reference Manual - Preliminary (Proprietary), Delphi Communications Corporation, El Segundo, Ca., April 1981.

[15] Hill, F.J., "Introducting AHPL," Computer, Vol. 7, No. 12, December 1972, pp. 28-30.

[16] Knudsen, M.H., "PMSL, An Interactive Language for System-Level Description and Analysis of Computer Structures", Carnegie-Mellon University, Pittsburgh, Pa., (NTIS AD 762 513), 1973.

[17] Lankskov, D., Davidson, S., Shriver, B., and Mallet, P.W., "Local Microcode Compaction Techniques," ACM Computing Surveys, Vol, 12, No. 3, September 1980, pp. 261-294.

[18] Ma, P. and Lewis, T., "On the Design of Microcode Compiler for a Machine-Independent High-Level Language," IEEE Transactions on Software Engineering, Vol SE-7, No. 3, May 1981, pp. 261-274.

[19] Marczynski, R.W. and Bakowski, P., "What Do the Computer Hardware Description Languages Describe?", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, October 1979.

[20] Maxey, G.F., and Organick, E.I., "CASL - A Language for Automating the Implementation of Computer Architectures," Proceedings of the 4th International Symposium on Computer Hardware Description Languages, October 1979.

[21] MC10800 Application Notes (AN-776 and AN-792), Motorola Semiconductor Products, Inc., Phoenix, Az., August 1977 and May 1979.

[22] MC68000 Design Module User's Guide (MEX68KDM-02), Motorola Semiconductor Products, Inc., Phoenix, Az., August 1979.

[23] MDAC Model 673 Processor Reference Manual (Proprietary), McDonnell Douglas Astronautics Company, Huntington Beach, Ca.

[24] Meinen, P., "Formal Semantic Description of Register Transfer Language Elements and Mechanized Simulator Construction," Proceedings of the 4th International Symposium on Computer Hardware Description Languages, October 1979.

[25] Microprogramming Handbook (AMD 2900), Advanced Micro Devices Corporation, Sunnyvale, Ca., 1976.

[26] MULTI Micromachine Description, Nanodata Corporation, Williams ville, N. Y.

[27] Nagle, A. W., "Automatic Synthesis of Microcontrollers," Proceedings of the 11th Annual Workshop on Microprogramming, November 1978.

[28] Nash, J. and Spak, M., "Hardware and Software Tools for the Development of a Micro-Programmed Microprocessor," Proceedings of the 12th Annual Workshop on Microprogramming, November 1979.

[29] Programmers Reference Manuals for the AP-120B - Part One and Part Two, Floating Point Systems, Inc., Beaverton, Oregon, January 1978.

[30] Schuler, D. M., "A Lauguage for Modeling the Functional and Timing Characteristics of Complex Digital Components for Logic Simulation," Proceedings of the 4th International Symposium on Computer Hardware Description Languages, October 1979.

[31] Siewiorek, D. P., "Introducing ISP," Computer, Vol. 7, No. 12, December 1972, pp. 39-41.

[32] Su,S., "Hardware Description Language Description: An Introduction and Prognosis," IEEE Computer, June 1977.

[33] Technical Report: Target Machine Description Methodology for HOL-to-Microcode Compilation - Preliminary Specification, JRS Industries, Inc., San Diego, Ca., March 1980.

[34] TRW 2-AU-80 Processor Operating Manual (TRW IOC 7334.4-187), TRW, Inc., Redondo Beach, Ca., February 1978.

[35] TRW 2-AU-80 Processor Specification (TRW IOC 7334.4-188), TRW, Inc., Redondo Beach, Ca., August 1978.

[36] VAX 11/780 Data Path Description Manual, Digital Equipment Corporation, Maynard, Mass., February, 1979.

[37] VAX 11/780 Microprogramming Tools, Digital Equipment Corporation, Maynard, Mass., July 1980.

JRS

# APPENDICES

JRS

APPENDIX A Published Paper:

# Automatic Microcode Generation for Horizontally Microprogrammed Processors

JRS

# APPENDIX B:

# CONTINUING STUDY AND EXPERIMENTATION PLANS

## 1.0 INTRODUCTION

To meet the needs discussed in Section 4.3 of this report, continued research and experimentation is required. That is, using the data and results generated and documented in this report as a baseline and utilizing an existing Automatic Microcode Generation System as an experimental 'testbed', continued research and experimentation is needed which is directed towards establishing the practicality and efficacy of the Microarchitecture Description Methodology (MDM) designed for use in the compilation process.

The effort would have as its primary objective the development and demonstration of a practical, generalized, and portable procedure to be used to retarget the compilation process; particular emphasis should be placed on the ease of use of the procedure and on the efficiency of the resulting microcode. The demonstration should also include 1) the use of the Dynamic Instruction Set Interpretation Techniques, 2) the use of Resource Allocation/Deallocation Techniques to aid in dynamic instruction set interpretation, and 3) the use of a generalized microcode generator in the compilation process.

A secondary objective of the effort would be an assessment of the feasibility of using the MDM to produce verification and validation tools to be applied to the microcode generated by the compilation process, as modified by the primary objective.

In summary, then, continued work is needed to develop the tools necessary to perform experimentation in the Microarchitecture Description Methodology area, to conduct experimentation aimed at testing the approaches taken, at generalizing to diverse microarchitectures and at uncovering and exploring technical issues of significance.

## 2.0 STUDY AND EXPERIMENTATION TOPICS

Using data and results generated under Contract DAAG29-81-C-0018 as a point of departure, continuing studies and experimentation should be directed toward the demonstration of the efficacy of a microarchitecture description methodology (MDM) designed for use in the automatic generation of microcode, from a HLL, for horizontally microprogrammed processors. In determining the efficacy of the MDM, the primary measure should be the efficiency of the microcode produced. Cost and ease of creating the MDM with sufficient descriptive capability should also be considered.

Efforts are also needed to study and evaluate resource allocation/deallocation techniques to aid in the use of the MDM to generate efficient microcode, the use of a more generalized microcode generator, and the use of the MDM to produce verification and validation tools for the generated microcode.

In the conduct of the continuing study and experimental effort, the following topics and tasks should be addressed:

- Continue to review and evaluate the microarchitectures of the horizontally microprogrammed processors available and select functionally diverse architectures to be utilized as the basis of the study and experimental effort.

- Continue to review the MDM previously developed and apply it to the microarchitectures selected above, taking into account the characteristics of the automatic microcode generation system and the special features and facilities of the selected microarchitectures.

- Enhance and make extensions to the MDM to accomplish instruction set interpretation in an efficient manner. This should focus on dynamic interpretation and should use the resource allocation/deallocation schemes and techniques developed below.

- Study, evaluate, and develop resource allocation/deallocation techniques, which take into account the detailed functions and facilities of the selected architectures and which make use of the information contained in the MDM, to aid in the generation of more efficient microcode.

JRS

- Further enhance and make extensions to the code generation portion of the HLL-to-microcode compilation system to allow for the use of the more generalized techniques developed above.

- Prepare and select appropriate applications as test cases.

- Prepare in HLL format and run test cases, of portions of the applications code selected above, through the compilation system, examine the results, and determine the effectiveness of the process.

- Attempt to optimize and improve the microcode generated through the use of the above mechanism by making modifications as required.

- Compare the microcode resulting from the 'new' compilation process to that manually generated to obtain a measure of efficiency of the compilation process and to acquire insights as to how to improve the process.

- Attempt to improve the efficiency of executing the entire compilation system, particularly in the MDM and code generation areas, by reviewing and analyzing the algorithms, techniques, and methodologies implemented looking for tradeoffs to reduce overall time, complexity and compilation costs.

- Further enchance and make extensions to the MDM to provide for the development of verification and validation tools. In particular, focus on the use of the MDM to provide a target microarchitecture simulator with which to verify and validate the generated microcode.

## 3.0 STUDY AND EXPERIMENTATION PLANS

The topics and tasks listed in the previous section represent a minimal continuing study and experimentation effort. The effort should be structured with enough flexibility to allow for the investigation and inclusion of additional topics which amy result from the pursuit of an indepth and thorough consideration of these topics. Moreover, the efforts specified in the list require an iterative approach; that is, continuous 'feed back' of the knowledge derived at each step is used to refine and enhance the techniques of the previous steps.

Several of the topics listed earlier are of significant interest to researchers in this field. In particular, technical treatise on the subjects of:

- Instruction set interpretation techniques,
- Resource allocation/deallocation techniques for microcode improvement and generation, and
- Microcode verification and validation techniques and methodologies

would be welcomed. Thus, as part of the continuing study and experimentation, the preparation, submittal, and presentation of papers to technical journals, publication, seminars, and conferences, as appropriate, to insure the widest possible dissemination of the findings of efforts in these and other allied areas should be planned.

For planning purposes, the topics of the previous section have been scheduled and are presented in that form below. This schedule is an attempt at organizing the activities presented earlier and is meant to be used only as a planning guide. The schedule implies a study and experimentation project approximately 15 calendar months in duration if the topics discussed in the previous section are used as a statement of the efforts to be performed.

It is anticipated that several topics, in particular microcode verification and validation considerations, will require more effort to define and finalize. However, investigations performed in this incremental manner will allow for the evaluation of progress and

the determination of the technical merit of the topic being developed, the analysis being performed or the study being conducted, and will allow for the dissemination and/or incorporation of current scientific and/or technical information. It also allows for an interim summary and status of all work done in all topics considered, including that yielding negative results or positive results not used as well as interim conclusions and recommendations for still further evaluation and/or redirection.

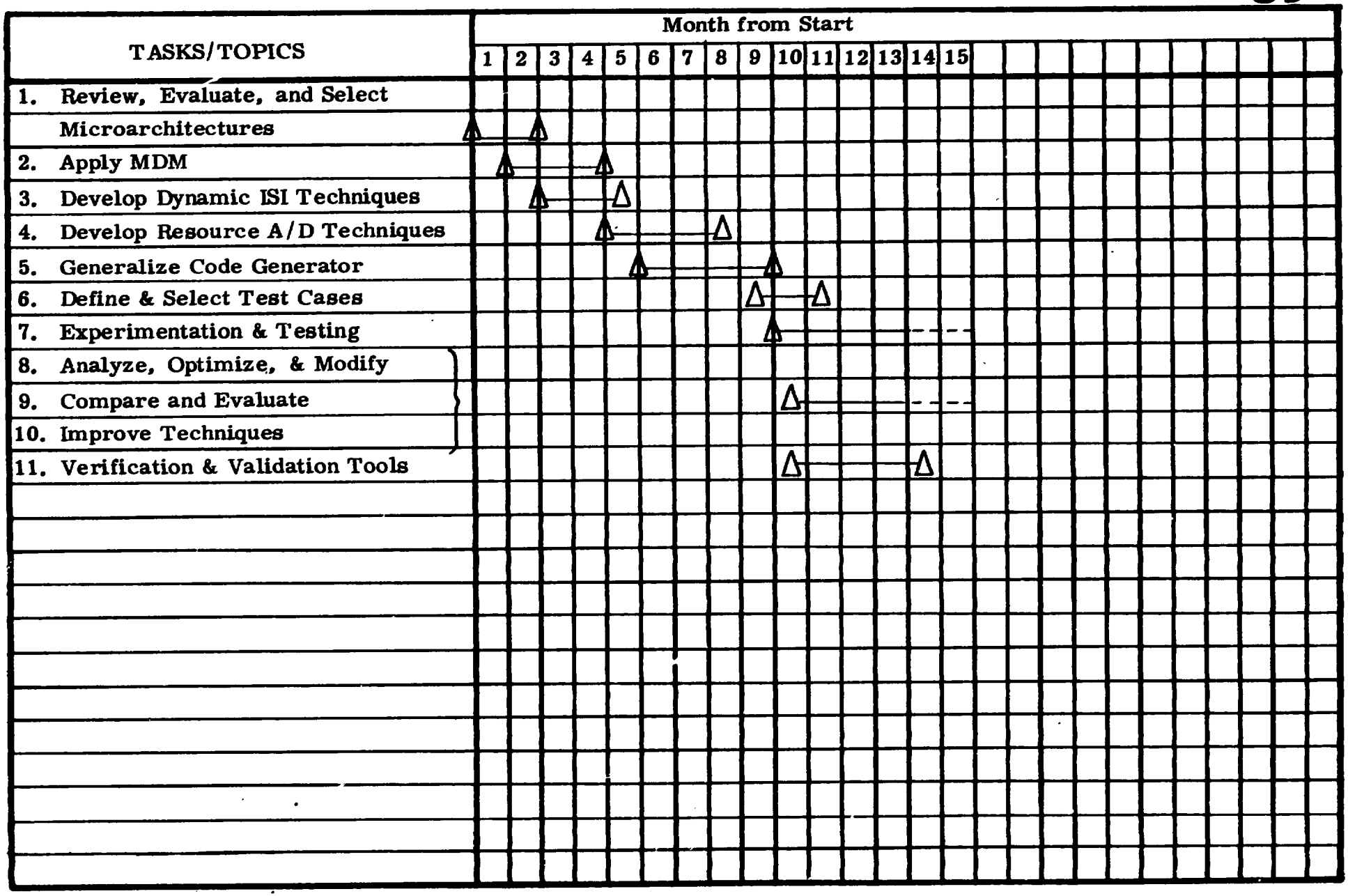
SCHEDULE



PROJECT/TASK NAME Microarchitecture Description Methodology Final Report - Appendix B

PREPARED BY J. Gieser CURRENT DATE 15 Oct 1981

JRS




Form No. JRS1009